C14

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/*
** EDMDispatchConfig.c
**
** Mission Statement: These are functions to deal with the config file
**                    APIs.
                              Since the config file can't be handled in a
**                    thread safe manner we need to mutex lock around
**                    all uses of it.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
** Basic idea here: Module for config file interaction
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispatchConfig.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <unistd.h>
#include <pthread.h>
#include <memory.h>
#include <sys/time.h>
#include <sys/types.h>

#include <ebconfig/rbconfig.h>
#include <EDMDispatchConfig.h>

static RBC_CONFIGS *rbc = NULL;

static boolean_ty  first = TRUE;

static pthread_mutex_t    G_configMtx = PTHREAD_MUTEX_INITIALIZER;

static void
DispatchInitializeConfigMutex()
{
   pthread_mutex_init(&G_configMtx, NULL);
   first = FALSE;
}
```

```
void
DispatchReadConfig()
{
    eperrno  ret;

    if (first == TRUE)
        DispatchInitializeConfigMutex();

    pthread_mutex_lock(&G_configMtx);

    if (rbc != NULL)
    {
        rbc_freeconfig(rbc);
        rbc = NULL;
    }

    ret = rbc_parse_config(NULL, &rbc,
                                   RBC_PARSE_DO_NOT_PRESERVE |
                                                    RBC_PARSE_APPLY);

    pthread_mutex_unlock(&G_configMtx);
}
```

```
boolean_ty
DispatchCheckRestorePermission(char *host, char *username)
{
    int         root = 0777;
    eperrno     err;
    boolean_ty allowed;

    // The Configuration structure is not set up correctly !!! STEVE
                                                               HOWARD

    pthread_mutex_lock(&G_configMtx);

    allowed = rbc_canirecover(rbc, host, username, &root, &err);

    pthread_mutex_unlock(&G_configMtx);

    allowed=1;
    return allowed;
}
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/*
** EDMDispatchBackground.c
**
** Mission Statement: This is the entry point for the cleanup thread.
**                    Its main purpose is to do some background
                                                        processing
**                    that we don't want to do elsewhere.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
** Basic idea here: Module for Background thread.
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(
                              #)$RCSfile: EDMDispatchBackground.c,v $ "
                             "$Revision: 1.23 $ "
                             "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <unistd.h>
#include <pthread.h>
#include <memory.h>
#include <sys/time.h>
#include <sys/types.h>

#include <restore/dispatch_daemon.h>
#include <EDMDispatchConfig.h>
#include <EDMDispatchSession.h>
#include <EDMTimedMessageApi.h>
#include <EDMDispatchBackground.h>

// Number of background activities run from the cleanup thread
#define NUMBER_OF_ACTIVITIES   4

static const int oneDaySeconds = SECONDS_PER_DAY;
                                                // one day in seconds
static const int configSeconds = 30; // 30 seconds
static const int oneMinute = 60; // one minute in seconds

// Structure used in the cleanup thread to schedule background
                                                        activities
struct Schedule {
    long  frequency;
    long  lastrun;
    long  nextrun;
    void  (*cleanupfunc)();
};
```

```
void *
DispatchBackground(void *buff)
{
    time_t    currTime;
    time_t    sleepfor = 0;
    time_t    difference = 0;

    // These are all the activities that are scheduled
    //    Frequency,        lastrun,     nextrun,     function to call
    struct Schedule  sched[NUMBER_OF_ACTIVITIES] = {
      //         {
    configSeconds,         -1,          -1,         DispatchReadConfig },
        { SECONDS_PER_HOUR,       -1,          -1,
                                                 CheckDispatchSessions },
        { oneMinute * 5,          -1,          -1,
                                             SendPingMessagesToSession },
        { oneMinute,              -1,          -1,
                                              DrainSessionDescriptors },
        { oneMinute,              -1,          -1,
                                               ReportLateTimedMessage }
    };

    //    DispatchReadConfig();

    // Initialize each elements last and next run.
    // The first run will be after sleepfor seconds.
    for (int i = 0; i < NUMBER_OF_ACTIVITIES; i++)
    {
        sched[i].lastrun = time(NULL);
        sched[i].nextrun = sched[i].lastrun + sched[i].frequency;

        difference = sched[i].nextrun - sched[i].lastrun;

        // We need to set the sleepfor value to something on the
        // first pass so we have something to compare to. The
        // lowest time is what we'll sleep for.
        if (i == 0 || difference < sleepfor)
            sleepfor = difference;
    }

    // Run things forever
    while(1)
    {
      // Sleep for the shortest amount of time needed
      sleep(sleepfor);

      currTime = time(NULL);

      // See which activities need to be run on this pass.
      for (int i = 0; i < NUMBER_OF_ACTIVITIES; i++)
      {
          if (sched[i].nextrun <= currTime)
          {
              // This activity needs running. Call the function
              // and change the lastrun and next run values.
              sched[i].cleanupfunc();
              sched[i].lastrun = currTime;
              sched[i].nextrun = sched[i].lastrun + sched[i].frequency;
          }

          // See how long until this needs to be run.
          difference = sched[i].nextrun - currTime;

          // We need to set the sleepfor value to something on the
          // first pass so we have something to compare to. The
          // lowest time is what we'll sleep for.
```

```
            if (i == 0 || difference < sleepfor)
                sleepfor = difference;
        }
    } // while (1)

    return buff;
}
```

```
/*
** Copyright 1996,1999 EMC Corporation
*/

/*
** EDMDispatchSession.cc
**
** Mission Statement: This is where all session management occurs.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      USE_SUNRPC - Compile source with sunrpc
                                                    support.  If
**                                  not set, assume DCE support.
**
** Basic idea here: Module for session management
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispatchSession.cc,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <pthread.h>
#include <memory.h>
#include <sys/time.h>
#include <sys/types.h>
#include <syslog.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/bintree.h>

#include <csc/cscomm.h>
#include <restore/dispatch_daemon.h>
#include <restore/dispatch_protocol_client.h>
#include <EDMSession.h>
#include <EDMReturnMessageApi.h>
#include <EDMDDHandleMgrApi.h>
#include <EDMDispatchSession.h>
#include <EDMDispatchConfig.h>
#include <EDMDDcr_rstsvc.h>

#include <EDMDispatchLog.h>

static RWBinaryTree       G_sessionTree;

static pthread_mutex_t    G_sessionTreeMtx = PTHREAD_MUTEX_INITIALIZER;
extern ELinkHandlePtr_ty  ELinkHandle;
```

```
static int maxDisconnectTime = SECONDS_PER_HOUR; // one hour

/*************************************************************************
                                                                 ********
**
** Routine:  LockSessionMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Lock the session mutex.
**
*************************************************************************
                                                                 ********
*/

static void
LockSessionMutex()
{
    static boolean_ty first = TRUE;

    if (first == TRUE)
    {
        first = FALSE;
        pthread_mutex_init(&G_sessionTreeMtx, NULL);
    }

    pthread_mutex_lock(&G_sessionTreeMtx);
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  UnlockSessionMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Unlock the mutex for the session tree object
**
***********************************************************************
                                                                ********
*/

static void
UnlockSessionMutex()
{
    pthread_mutex_unlock(&G_sessionTreeMtx);
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  InitializeSession
**
** Inputs:   DD_initialize_args *arg - args sent via RPC for starting
                                                              session
**           struct svc_req *req - the request block from RPC
**
** Outputs:  DD_initialize_result *res - the result structure which
                                                          tells whether
**                   operation succeeded or failed.
**
** Return Codes:
**           None
**
** Purpose:  Initialize a session for the GUI.
**
***********************************************************************
                                                                ********
*/

void
InitializeSession(IN DD_initialize_args *arg, IN struct svc_req *req,
                  OUT DD_initialize_result *res)
{
    EDMSession  *session;
    EDMSession  *ret;
    pthread_t   id;
    time_t      t;

    if (arg == NULL || req == NULL || res == NULL)
    {
        return;
    }

    t = time(NULL);

    session = new EDMSession();

    if (session == NULL)
    {
        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        return;
    }

    session -> initSession();

    session -> setStartTime(t);

    session -> setOperationType(arg -> service);

    session -> setStatus(DD_SERVICE_STARTING);

    if (arg  > username  != NULL && arg -> hostname != NULL)
    {
      switch(arg -> service)
        {
           // code is commented out because we do not
           // want to read tthe config for permission information
           // at this time, it is a waste of cycles
#if 0
             case DD_SERVICE_RESTORE : boolean_ty allowed;

                                  allowed =
                                        DispatchCheckRestorePermission(
                                            arg->hostname,
                                                   arg -> username);

                                    if (!allowed)
                                    {
                                        res -> status =
                                             DD_SERVICE_FAILURE_PERMS;
                                       delete session;
                                        return;
                                    }

                                    break;
#endif
        default: // Add some error messsage for unknown service
                                    break;
        };
    }
    else
    {
        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        delete session;
        return;
    }

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.insert((
                                        RWCollectable *) session);

    UnlockSessionMutex();

    if (ret == NULL)
    {
        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        delete session;
        return;
    }

    session -> getSessionID(&res -> service_handle);

    // Call Steve's thread
    pthread_create(&id, NULL, &DDRSTsvc_init, (void *) session);

    session -> setThreadID(id);

    return;
}
```

```
/**********************************************************************
                                                                ********
**
** Routine:  SendPingMessagesToSession
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Queue up all the ping messages to the sessions.
                                                         If they don't
**           respond they should be considered dead.
**
**********************************************************************
                                                                ********
*/

void
SendPingMessagesToSession()
{
    EDMSession *sess;

    LockSessionMutex();

    RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                       G_sessionTree);

    while ( sessionIterator != NULL &&
           (sess = (EDMSession*) (*sessionIterator)()) != NULL )
    {
        DD_client_session_id sid;
        rpc_binding_handle_t *cscb = NULL;
        int                     status;
        int                     ret;

        if (sess -> getStatus() != DD_SERVICE_RUNNING)
            continue;

        sess -> getSessionID(&sid);

        ret = GetCSCHandle(&sid, &cscb, &status);

        if (ret != 0 || cscb == NULL || *cscb == NULL)
            continue;

        PushResponseMessage(dp_ping_request, sid, cscb, &status);
    }

    // through with iterator
    if (sessionIterator != NULL)
    {
        delete sessionIterator;
    }

    UnlockSessionMutex();
}
```

```
/***********************************************************************
                                                               ********
**
** Routine:  UpdateSessionLastReceived
**
** Inputs:   DD_client_session_id *sessID - session that sent us
                                                             something
**
** Outputs:  None
**
** Return Codes:
**           0 on success and non-zero otherwise
**
** Purpose:  Update the specified session with the lastest received
                                                               message
**           time.
**
***********************************************************************
                                                              ********

*/

int
UpdateSessionLastReceived(DD_client_session_id *sessID)
{
    time_t   last = time(NULL);
    EDMSession  *session;
    EDMSession  *ret;

    session = new EDMSession();

    if (session == NULL)
    {
        EDMDispatch_logent(
                  __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                         "Failure to create a session block");
        return -1;
    }

    session -> setSessionID(sessID);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

    UnlockSessionMutex();

    delete session;

    if (ret == NULL)
    {
        EDMDispatch_logent(
            __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                         "Failure to update session %ld:%ld received
                                                               time",
                         sessID -> high, sessID -> low);
        return  1;
    }

    ret -> setLastReceived(last);

    return 0;
}
```

```
/***********************************************************************
                                                               ********
**
** Routine:  UpdateSessionLastSent
**
** Inputs:   DD_client_session_id *sessID - session that sent us
                                                             something
**
** Outputs:  None
**
** Return Codes:
**           0 on success and non-zero otherwise
**
** Purpose:  Update the specified session with the lastest sent
                                                               message
**           time.
**
***********************************************************************
                                                              ********

*/

int
UpdateSessionLastSent(DD_client_session_id *sessID)
{
    time_t   last = time(NULL);
    EDMSession  *session;
    EDMSession  *ret;

    session = new EDMSession();

    if (session == NULL)
    {
        EDMDispatch_logent(
                  __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                         "Failure to create a session block");
        return -1;
    }

    session -> setSessionID(sessID);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

    UnlockSessionMutex();

    delete session;

    if (ret == NULL)
    {
        EDMDispatch_logent(
            __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                         "Failure to update session %ld:%ld sent
                                                               time",
                         sessID -> high, sessID -> low);
        return -1;
    }

    ret -> setLastSent(last);

    return 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  CheckDispatchSessions
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Look for dead sessions and kill them off
**
***********************************************************************
                                                               ********
*/

void
CheckDispatchSessions()
{
    EDMSession   *sess;
    int           status = 0;
    int           ret = 0;
    time_t        currTime;
    RWBinaryTree reaperTree;

    currTime = time(NULL);

    LockSessionMutex();

    RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                   G_sessionTree);

    while ( sessionIterator != NULL &&
            (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {

        if ( (sess->getLastReceived(
) <= currTime - maxDisconnectTime && sess->getLastReceived() != 0) ||
             (sess->getStartTime() <= currTime - maxDisconnectTime &&
             (sess -> getStatus(
                ) == DD_SERVICE_FAILURE_NONEXEC || sess -> getStatus(
                                    ) == DD_SERVICE_FAILURE_EXEC ||
              sess -> getStatus() == DD_SERVICE_FAILURE_PERMS)) )
        {
            // Insert it into the reaper tree
            (void) reaperTree.insert(sess);
        }
    }

    // through with iterator
    if (sessionIterator != NULL)
    {
        delete sessionIterator;
    }

    UnlockSessionMutex();

    // If the reaper tree has something in it then use those entries
                                                           to remove
    // things from the query tree.
    if (reaperTree.entries() > 0)
    {
        sessionIterator = new RWBinaryTreeIterator(reaperTree);
```

```
        while ( sessionIterator != NULL &&
            (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {
            DD_client_session_id  sessID;

            sess -> getSessionID(&sessID);

            ret = removeSession(&sessID, &status);

            if (ret != 0)
            {
                EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
                                "Failure to remove session %ld:%ld",
                                sessID.high, sessID.low);
                continue;
            }
            else
            {
                EDMDispatch_logent(
                                __FILE__, __LINE__, LOG_INFO, 0, 0,
                        "Removing session %ld:%ld,
                Haven't recieved anything since %ld. Current %ld",
                        sessID.high, sessID.low,
                                sess -> getLastReceived(),
                        currTime - maxDisconnectTime);
            }

            ret = deleteHandleSet(&sessID, &ELinkHandle, &status);

            if (ret != 0)
            {
                EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
                                "Failure to delete handles for
                                            session %ld:%ld",
                                sessID.high, sessID.low);
            }
        }

        // through with iterator
        if (sessionIterator != NULL)
        {
            delete sessionIterator;
        }

        reaperTree.clear();
    }
}
```

```
/**********************************************************************
                                                               ********
**
** Routine:  DrainSessionDescriptors
**
** Inputs:    None
**
** Outputs:  None
**
** Return Codes:
               None
**
** Purpose:  Drain whatever data is on stdout and stderr for sessions.
**
**********************************************************************
                                                              ********
*/

void
DrainSessionDescriptors()
{
    int      hout = 0, herr = 0, status = 0;
    int      selret = 0;
    int      i = 0;
    char     buff[1024];
    struct timeval timetowait = {
           1, 0
           };
    fd_set  stdoutSet;
    fd_set  stderrSet;

    getStdoutSet(&stdoutSet, &hout, &status);

    if ( (selret = select(
                hout + 1, &stdoutSet, NULL, NULL, &timetowait)) >= 0)
    {
        for (; i < hout+1; i++)
        {
            if (FD_ISSET(i, &stdoutSet))
            {
                while (read(i, buff, 1024) > 0);
            }
        }
    }

    getStderrSet(&stderrSet, &herr, &status);

    if ( (selret = select(
                herr + 1, &stderrSet, NULL, NULL, &timetowait)) >= 0)
    {
        for (i = 0; i < herr+1; i++)
        {
            if (FD_ISSET(i, &stderrSet))
            {
                while (read(i, buff, 1024) > 0);
            }
        }
    }
}
```

```
/**********************************************************************
                                                               ********
**
** Routine:  GetSessionStatus
**
** Inputs:   DD_client_session_id *ssid - session ID to check the
                                                          status of
**
** Outputs:  int *status - status of the function call
**           int *s_status - session status
**
** Return Codes:
               0 if successful and non-zero otherwise
**
** Purpose:  Get status on the session.
**
**********************************************************************
                                                              ********
*/
int
GetSessionStatus(
               DD_client_session_id *ssid, int *s_status, int *status)
{
    EDMSession  *sess;
    EDMSession  *ret;

    if (status == NULL)
    {
        return -1;
    }

    if (ssid == NULL || s_status == NULL)
    {
        *status = SESSION_BAD_ARGS;

        return -1;
    }

    sess = new EDMSession();

    if (sess == NULL)
    {
        EDMDispatch_logent(
                  __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                        "Failure to create a session block");
        *status = SESSION_NO_MEMORY;

        return -1;
    }

    sess -> setSessionID(ssid);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

    UnlockSessionMutex();

    delete sess;

    if (ret == NULL)
    {
        EDMDispatch_logent(
              __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                        "Failure to lookup session %ld:%ld",
                                ssid -> high, ssid -> low);
        *status = SESSION_LOOKUP_FAILED;
        return -1;
    }

    *s_status = ret -> getStatus();

    return 0;
}
```

```
/************************************************************************
********
**
** Routine:  GetDispatchStatus
**
** Inputs:   DD_getservicestatus_args *arg - session ID to check the
                                                              status of
**
** Outputs:  DD_getservicestatus_result *res - the result structure
                                                              which tells
**                    whether operation succeeded or failed.
**
** Return Codes:
**            None
**
** Purpose:  Get status on the starting session.
**
*************************************************************************
********

*/

void
GetDispatchStatus(IN DD_getservicestatus_args *arg,
                  OUT DD_getservicestatus_result *res)
{
    EDMSession  *sess;
    EDMSession  *ret;
    static char buff[CONNECT_HANDLE_SIZE];

    sess = new EDMSession();

    if (sess == NULL)
    { // Give an error
        EDMDispatch_logent(
                  __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                           "Failure to create a session block");
        return;
    }

    sess -> setSessionID(&arg -> service_handle);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

    UnlockSessionMutex();

    delete sess;

    if (ret == NULL)
    {
        EDMDispatch_logent(
                  __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                           "Failure to lookup session %ld:%ld",
                           arg -> service_handle.high,
                                       arg -> service_handle.low);

        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        return;
    }

    res -> status = ret -> getStatus();

    memset(buff, 0, sizeof(buff));
    if (res -> status == DD_SERVICE_RUNNING)
    {
        res -> handle.handle_val = (char *) ret -> getConnectionHandle(
                                                                      );
        res -> handle.handle_len = CONNECT_HANDLE_SIZE;
    }
    else
    {
        res -> handle.handle_val = (char *) buff;
        res -> handle.handle_len = CONNECT_HANDLE_SIZE;
    }
}
```

```
/************************************************************************
                                                                ********
**
** Routine:  GetDispatchInfo
**
** Inputs:   DD_getservicestatus_args *arg - session ID to check the
                                                           status of
**
** Outputs:  SessionBlock *res - the information regarding the
                                                    specified session
**
** Return Codes:
**           None
**
** Purpose:  Get status on all the sessions.
**
************************************************************************
                                                               ********
*/

void
GetDispatchInfo(IN DD_getservicestatus_args *arg,
                OUT SessionBlock *res)
{
    EDMSession  *sess;
    EDMSession  *ret;
    SessionInfo  *sinfo, *slast;
    static char buff[CONNECT_HANDLE_SIZE];

    LockSessionMutex();

    if (arg -> service_handle.high != 0 && arg -> service_handle.
                                                       low != 0)
    {
        // Looking for a single session. Do a find.
        sess = new EDMSession();

        if (sess == NULL)
        { // Give an error
            EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                                 "Failure to create a session block");

            UnlockSessionMutex();
            return;
        }

        sess -> setSessionID(&arg -> service_handle);

        ret = (EDMSession *) G_sessionTree.find(sess);

        delete sess;

        if (ret == NULL)
        {
            EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                               "Failure to lookup session %ld:%ld",
                          arg -> service_handle.high,
                                     arg -> service_handle.low);
            UnlockSessionMutex();
            return;
        }

        res -> totalsessions = 1;
        res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

        if (res -> sess == NULL)
        {
            EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                              "Failure to allocate session info
                                                          block");
            UnlockSessionMutex();
            return;
        }

        sinfo = res -> sess;

        ret -> getSessionID(&sinfo -> service_handle);
        sinfo -> status = ret -> getStatus();
        sinfo -> jobstarttime = ret -> getStartTime();
        sinfo -> operation_type = ret -> getOperationType();
        sinfo -> lastSent = ret -> getLastSent();
        sinfo -> lastReceived = ret -> getLastReceived();
    }
    else
    {
        res -> totalsessions = 0;

        res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

        if (res -> sess == NULL)
        {
            EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                              "Failure to allocate session info
                                                          block");
            UnlockSessionMutex();
            return;
        }

        sinfo = res -> sess;

        RWBinaryTreeIterator *sessionIterator = new
                                RWBinaryTreeIterator(G_sessionTree);

        boolean_ty addnext = FALSE;

        while ( sessionIterator != NULL && (ret = (EDMSession*) (
                                   *sessionIterator)()) != NULL )
        {
            int                  status;

            if (addnext)
            {
                sinfo -> next = (SessionInfo *) calloc(1, sizeof(
                                                    SessionInfo));

                if (sinfo -> next == NULL)
                {
                    break;
                }

                sinfo = sinfo -> next;
            }

            ret -> getSessionID(&sinfo -> service_handle);
            sinfo -> status = ret -> getStatus();
            sinfo -> jobstarttime = ret -> getStartTime();
```

```
            sinfo -> operation_type = ret -> getOperationType();
            sinfo -> lastSent = ret -> getLastSent();
            sinfo -> lastReceived = ret -> getLastReceived();

            getHandleSet(
                        &sinfo -> service_handle, &sinfo -> outhandle,
                         &sinfo -> errhandle, &status);

            res -> totalsessions++;

            sinfo -> next = NULL;
            addnext = TRUE;
        }

        // through with iterator
        if (sessionIterator != NULL)
        {
            delete sessionIterator;
        }

    }

    UnlockSessionMutex();
}
```

```
/************************************************************************
                                                                 ********
**
** Routine:  removeSession
**
** Inputs:
**
** Outputs:
**
** Return Codes:
**              None
**
** Purpose: Remove the active session object between the GUI and the
                                                                 Service.
**
************************************************************************
                                                                 ********
*/

int
removeSession(IN DD_client_session_id *sess_id,
              OUT int *status)
{
    EDMSession *sess;
    EDMSession *ret;

    if (status == NULL)
    {
        return -1;
    }

    if (sess_id == NULL)
    {
        *status = SESSION_BAD_ARGS;
        return -1;
    }

    *status = 0;
    if (G_sessionTree.isEmpty())
    {
        EDMDispatch_logent(
                 __FILE__, __LINE__, LOG_ERR, SESSION_LIST_EMPTY, 0,
                          "No sessions in list.
                                 Can't remove session <%ld:%ld>",
                          sess_id -> high, sess_id -> low);
        *status - SESSION_LIST_EMPTY;
        return -1;
    }

    sess = new EDMSession();

    if (sess == NULL)
    {
        EDMDispatch_logent(
                 __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                          "Failure to create a session block");
        *status = SESSION_NO_MEMORY;
        return -1;
    }

    sess -> setSessionID(sess_id);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.remove(sess);
```

```
    UnlockSessionMutex();

    if (ret == NULL)
    {
        EDMDispatch_logent(
                 __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                          "Failure to remove session %ld:%ld",
                          sess_id -> high, sess_id -> low);
        delete sess;
        *status = SESSION_LOOKUP_FAILED;
        return -1;
    }

    delete ret;
    delete sess;

    return 0;
}
```

```
/*
** Copyright 1996,1998 EMC Corporation
*/

/*
** EDMDispatchLog.c
**
** Mission Statement: This is the logging wrapper around esl logging.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
** Basic idea here: Module for logging
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: bamlogging.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <stdio.h>
#include <pthread.h>
#include <stdarg.h>
#include <string.h>

#include <logging/logging.h>
#include <EDMmain.h>
#include <EDMDispatchLog.h>

/*********************************************************************
                                                            ********
**
** Routine: EDMDispatch_logent
**
** Inputs:        file name       - c source file name
**                line number     - line number in c source file
**                priority        - esl logent priority
**                message #       - esl logent message number
**                errno           - errno optional
**                msg format      - format for printing,
                                          ie "text %s %d..."
**                variable args   - variable arguments for printing
**
** Outputs:  Calls esl_logent for output.
**
** Return Codes:
**               None
**
** Purpose:      All messages issued from the bamd should call this
                                                            routine
**                so that they can be consistently formatted.  It is
**                anticipated that this routine will also be used for
**                statistics gathering on messages and perhaps even
**                debugging.
**
** Intended caller:  internal only.
```

```
**
*************************************************************************
                                                                ********
*/

void
EDMDispatch_logent( IN char      *filename,
                IN int       linenum,
                IN int       priority,       /* esl_logent priority */
                IN int       msg_no,
                                          /* esl_logent message number */
                IN int       err_no,                /* errno */
                IN char      *msg,
                                          /* msg format for sprintf... */
                IN ...
                    )                    /* Remaining arguments to msg */
{

#define EDMDispatch_MSGBUFF 2048

char      tmpbuff[ EDMDispatch_MSGBUFF ];
char      msgbuff[ EDMDispatch_MSGBUFF ];
va_list  vap;                          /* Variable argument pointer */
static  pthread_mutex_t     log_mutex = PTHREAD_MUTEX_INITIALIZER;
static  boolean_ty          first = TRUE;

  if (first)
  {
      first=FALSE;
      pthread_mutex_init(&log_mutex, NULL);
  }

  if (msg == NULL || filename == NULL)
      return;

  /* Setup variable argument list processing */

  va_start(vap, msg);

  tmpbuff[ 0 ] = 0;
  msgbuff[ 0 ] = 0;


  /* Build caller message with all arguments. */

  (void) vsprintf( tmpbuff, msg, vap );

  if ( err_no > 0 )
  {
    char *str = strerror(err_no);

    if (str == NULL)
        str = "Unknown Error";

    (void) sprintf(      msgbuff,
                         "(%s-%d) %s %s <%d>",
                         filename,
                         linenum,
                         tmpbuff,
                         str,
                         err_no );
  }
  else
    (void) sprintf(      msgbuff,
                         "(%s-%d) %s",
                         filename,
                                linenum,
                                tmpbuff );

   /*
    * Use a mutex lock because esl_logent is NOT thread safe.
    */
   pthread_mutex_lock(&log_mutex);

   (void) esl_logent( priority, EB, EDMDISPATCH, msg_no, msgbuff );

   pthread_mutex_unlock(&log_mutex);
 } /* End of EDMDispatch_logent() */
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/


/* EDMDispatchService.c
 *
 *
 * Mission Statement:  RPC entry points.
 *
 * Primary Data Acted On:
 *
 * Compile-Time Options:
 *
 * Basic idea here:
*/

#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispatchService.c,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/inout.h>

#include <logging/logging.h>
#include <csc/cscomm.h>

#include <restore/csc_EDMDispatch.h>
#include <restore/dispatch_daemon.h>

#include <EDMDispatchLog.h>
#include <EDMDispatchSession.h>

/*
 * These are all the rpc entry points for the dispatch daemon.
 * The dispatch daemon is multi-threaded and it is the main thread
 * which handles all incoming RPC. ONC RPC is single threaded
 * so each call blocks other RPC calls. This provides us some
 * safety in the way we handle our data and limits our exposure
 * to unexpected multithreading problems.
 */
static void FreeSessionInfo(SessionInfo *);

/*********************************************************************
                                                             ********
**
** Routine:  dd_initialize_1
**
** Inputs:   DD_initialize_args * - args for the restore initialize
                                                                call
**
** Outputs:  None
**
** Return Codes:
**             DD_initialize_result * - result of init function call
**
** Purpose:  Function to create a restore session.
**
** Intended caller:  Internal Only.
*********************************************************************
                                                             ********
*/

DD_initialize_result *
```

```
dd_initialize_1_svc(
                    IN DD_initialize_args *arg, IN struct svc_req *req )
{
    static DD_initialize_result argzz;

    InitializeSession(arg, req, &argzz);

    return &argzz;
}
```

```
/*********************************************************************
                                                             ********
**
** Routine:  dd_getservicestatus_1
**
** Inputs:   DD_getservicestatus_args * - args for the
                                          getservicestatus call
**
** Outputs:  None
**
** Return Codes:
**              DD_getservicestatus_result * - result of status function
                                                                   call
**
** Purpose:  Function to poll for status on a session.
**
** Intended caller:  Internal Only.
*********************************************************************
                                                             ********
*/

DD_getservicestatus_result *
dd_getservicestatus_1_svc(
            IN DD_getservicestatus_args *arg, IN struct svc_req *req )
{
    static DD_getservicestatus_result argzz;

    GetDispatchStatus(arg, &argzz);

    return &argzz;
}
```

```
/*********************************************************************
                                                             ********
**
** Routine:  dd_getsessioninfo_1
**
** Inputs:   DD_getservicestatus_args * - args for the getsessioninfo
                                                                 call
**
** Outputs:  None
**
** Return Codes:
**              SessionBlock * - result of session info call
**
** Purpose:  Function to get information on all sessions.
**
** Intended caller:  Internal Only.
*********************************************************************
                                                             ********
*/

SessionBlock *
dd_getsessioninfo_1_svc(
            IN DD_getservicestatus_args *arg, IN struct svc_req *req )
{
    static SessionBlock argzz;
    static boolean_ty first = TRUE;

    if (first)
    {
        memset(&argzz, 0, sizeof(argzz));
        first = FALSE;
    }
    else
    {
        FreeSessionInfo(argzz.sess);
        argzz.sess = NULL;
    }

    GetDispatchInfo(arg, &argzz);

    return &argzz;
}
```

```
/***********************************************************************
                                                              ********
**
** Routine:  FreeSessionInfo
**
** Inputs:   SessionInfo * - arg to free
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Function to free all SessionInfo structures in a list.
**
** Intended caller:  Internal Only.
***********************************************************************
                                                            *********/

static void FreeSessionInfo(SessionInfo *sess)
{
    if (sess == NULL)
        return;

    if (sess -> next != NULL)
        FreeSessionInfo(sess -> next);

    free(sess);
}
```